An evaluation of the Ten15 persistent store

Tim Blanchard[†*]

John A. McDermid[‡]

†Glossa,
59 Alexandra Road,
Reading,
Berkshire
RG1 5PG
‡Department of Computer Science,
University of York,
Heslington,
York.
YO1 5DD

glossa@cix.compulink.co.uk
jam@minster.york.ac.uk

ABSTRACT

Persistent programming languages provide abstract mechanisms which completely describe the storage and retrieval of long term data. In this paper, the Ten15 persistent programming language is reviewed. Ten15 can be categorised as supporting a minimal persistence mechanism and an extremely powerful and expressive type system. Details of using Ten15 to build a general purpose filing system, based on dictionaries, are given. It is concluded that the ability to build a complex and flexible file system depends on two factors: the constructive power of the type system and the flexibility of the minimal persistent store.

* Research is sponsored by SERC/UK research grant number 89556293 and with the support of the Procurement Executive Ministry of Defence.

1. Introduction

Persistence is the ability for program data to endure longer than the invocation of the program which creates it [2]. It has formed the basis for a large research community which has identified the underlying principles of persistence and led to the construction of a number of persistent programming languages [1, 19, 22, 24, 27]. The resulting languages have been diverse, conforming to no common design principle for the paradigm. Thus we have languages ranging from the general purpose PS-algol [15], to the capability architecture of χ [18] and the functional programming language Staple [21]. Thus it seems that persistence is a principle which can be integrated into a programming paradigm, in addition to being a language class in its own right.

In this paper we address the relationship between persistent programming languages and operating systems [4]. An operating system is a nucleus of resources on which high level computing applications, such as programming and data processing, is based. An operating system provides facilities to store data in a secure and consistent manner; manage data in main memory; provide concurrent execution of processes; and schedule processes and computations [20].

Persistent programming languages provide the basic capabilities to support the main- and backing store management of data. The integrity of data and the maintenance of concurrent update is associated with these capabilities. Increasingly, other facilities traditionally expected of an underlying operating system, such as network management, device controllers and concurrency are being provided as base primitives of persistent programming languages [26].

In this paper we look at the abstract programming machine, Ten15 [7, 17], explore the capabilities of its persistent programming mechanism, and identify how secure filestore systems can be introduced into the language.

2. Ten15

Ten15 is the result of many years' research at the Defence Research Agency (formerly the Royal Signals and Radar Establishment), Malvern, UK into the development of a strongly typed intermediate language. The design rationale of Ten15 is to provide an algebraic basis for software development. This leads to a mechanism by which different computers can be made compatible through the implementation of the Ten15 algebra on each platform. In addition, Ten15 is a complete systems programming language, permitting the fine-grained allocation of mainstore, filestore and network resources. The facilities of the language extend its application area to fields such as high integrity systems, secure systems, IPSE development, heterogeneous networks and fine-grained databases. This is associated with a mathematically described strong type system, which ensures the type security of the system functions. The use of Ten15 means that its users can preserve their investment in existing software because Ten15 can co-exist along-side existing operating systems and, since it is an intermediate language for the majority of modern programming languages, such as Ada, Pascal, and Standard ML, it can provide a mechanism for mixed-language working.

Ten15 possesses an extremely flexible and expressive type system. It has a number of type system mechanisms which give the language great expressive power, such as first class closure, lambda type expressions, existential quantification, dynamic extended unions and typed data.

For this paper, one feature of Ten15 is of particular interest. That is the persistence

mechanism. It is based on a mechanism used in the KeepSake database kernel [23], and derived from experiences with the filestore of the Flex PSE [12]. The persistent filestore is called the *datastore*, into which arbitrarily complex Ten15 data fragments can be made persistent and preserved between program executions. The datastore is novel in that it is *immutable*, or non-overwritable. The datastore is structured hierarchically, radiating down from a single root. New items of persistent data are written from the centre of the disk outwards; since values cannot be overwritten, the update of a persistent value is equivalent to creating a new item of data. Since the disk pointer to the update is no longer valid, the filestore provides an indexing mechanism which can trace the changes in location of persistent data so that referential transparency can be maintained.

The root of the datastore is the only location that is overwritable. The root is protected by a four block, three stage commit algorithm to preserve its integrity in the face of error or corruption.

There are several advantages to immutable filestores over the more traditional overwritable ones. First, it makes the filestore more resilient to corruption from a premature commitment due to a crash or an error. This is because, until the root is assigned with a reference to the new state of the datastore, the old database is valid. If a crash occurs whilst the root is being written, the subsequent system initialisation will determine whether the update operation on the root was completed, since the root is not assigned unless the three protection blocks contain the same value. A second advantage is that the history of changes to the filestore since the last garbage collection is preserved in the immutable store, allowing privileged software to undo the changes to persistent variables. Thirdly, it provides a many readers, single writer mechanism for concurrent update, since any number of processes can access data through the original root while a process is preparing a modification ready for assignment to the root.

Ten15 provides a set of strongly typed operations which allow the datastore to be manipulated within the language without endangering the integrity of the system.

2.1. Persist and UnPersist

Data is explicitly preserved in the datastore by using a Persist operator provided within the Ten15 programming language, called the Ten15 notation. When Persist is invoked on an arbitrary piece of data, the data is compacted into a persistent format by tracing and concatenating all objects to which the data refers, altering all pointers and offsets within the concatenated block to reflect their new status as datastore objects, and writing the block onto a new location in the data store. A reverse operation, UnPersist, takes a persistent reference and delivers the mainstore representation of the persistent value, by copying the persistent block back into mainstore, updating the pointers within the block to their new absolute location in mainstore. This approach to persistence is an example of *replicating persistence* [3], and found in PPLs such as Amber [9] and χ [18], since repeated retrieval of the same persistent object causes identical copies of the object to be brought into mainstore. This is featured in figure 1 where a value is made persistent in a datastore, then twice brought back into memory. Operations affecting the one reference do not alter the other. In this case, the reference to the mainstore version of the persistent value is incremented and reassigned to the reference.

```
Let ds : datastore = DISK CAPABILITY
Let value : Int = 5
Let persistent_ref = Persist{value, ds}
Var r1 := UnPersist persistent_ref
```

```
Var r2 := UnPersist persistent_ref
In
  r1 := @ r1 + 1
Ni
```

Figure 1: Using Persist and UnPersist

2.2. Persistent variables

Persist and UnPersist provide a means of writing and reading a value to and from persistent storage. Due to the nature of the immutable storage, updating of a persistent value cannot occur. As a result of this, the Ten15 datastore supports *persistent variables* (pvars) which control the updating of single instances of persistent values. The mechanism for managing updates to persistent variables is to create a copy of the datastore. This is performed by updating the root of the datastore to refer to a new datastore where the persistent variable has been updated to reflect the change in its persistent value. The hierarchical structure of the datastore has also been altered to propagate the changes.

Persistent variables provide a means by which object identity of persistent objects can be preserved. The pvar acts as an outwardly unchanging reference to an object on the persistent store, since changes to the pvar are propagated to all references. Referential integrity is preserved by path following of updates and locations of the pvar, with the final update being the current value for the pvar object. Figure 2 is a sample Ten15 text for updating a persistent variable in place. Object identity is demonstrated because when this piece of code is executed, true will be delivered, despite the assignment of a new value to the pvar. This shows that it is the contents of a pvar which is updated rather than the variable reference itself.

```
Let root = DISK CAPABILITY
Let aPset = CreatePset root
Let pers = Persist{CrInt 5, ds}
Let pvar1 = CreatePvar{pers,
          aPset {position in datastore hierarchy}}
Let pvar2 = pvar1
In
     AssignPvar{pvar2,Persist{CrInt 6,ds}};
     If pvar1 = pvar2
     Then True
     Else False
     Fi
Ni
```

Figure 2: Demonstrating persistent object identity.

2.3. Persistent Sets

A hierarchical structuring mechanism can be used within a Ten15 datastore. The data structure for this is the *persistent set* (pset). If a datastore were flat, eventually as more pvars are added to the datastore, the data structure handling the pvars would become unwieldly and inefficient. Hence Ten15 supplies a mechanism for decomposing pvars into a tree of logically related pvars and psets, where only the group of pvars and all the groups of psets and pvars directly leading to the root of the datastore are updated when a pvar is reassigned. This is displayed in figure 3, where a persistent value is updated and a hierarchical structure of pvars and psets is updated as necessary to reflect the new nature of the datastore. Ten15 code which demonstrates the use of psets can be found in figure 2, where a pset *aPset* is created as a child of the root pset using *CreatePset*.

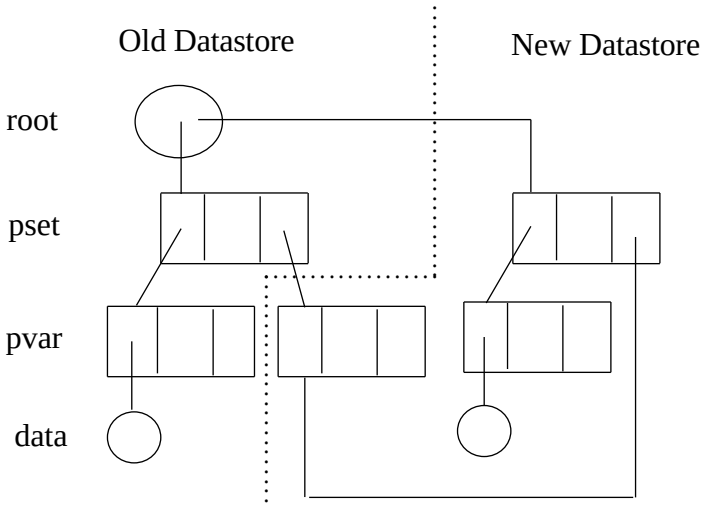

Figure 3: Updating a persistent value in a datastore.

2.4. Garbage Collection

Since the datastore is constantly being written to and no parts of it are ever re-used, there comes a time when the datastore becomes full. As a result of this, a garbage collection algorithm is called which reclaim the space occupied by the root of the datastore. A similar event must take place in the mainstore to clear out all data that can no longer be reached from the existing execution display. An efficient garbage collector for both the mainstore and datastore is central to the success of Ten15, since both resources are hidden from the programmer so he cannot exercise discretion in allocation in order to avoid the need for the release of memory or datastore that are no longer required.

2.5. Guaranteeing consistency

The Ten15 datastore is updateable at a single location, the root. The root refers to a pset which holds all the details of the filestore, such as other psets and pvars. The root is itself a pset. As the hierarchy of psets is updated, the root must itself be rewritten with the new disk address of the pset.

Since the root is writeable, it is prone to be corrupted due to program or hardware failure.

However, the root is protected by a four block, three stage commit algorithm. The root is divided into three sections: one contains two blocks which hold two shadow copies of the root pointer; another is another shadow copy; and there is the root copy itself. The update procedure is to write the new root into the two root block, then into the second block. Finally the root block is overwritten with the new value.

If a crash occurs, the root update algorithm allows it to be detected and for the old state of the filestore to be reconstructed if necessary. A filestore is consistent if all four blocks hold the same value. If a crash occurs whilst or after writing the first two blocks then the value that they hold will differ from one to the other and from the third block, and the old state of the filestore is restored. If it occurs whilst writing the third or fourth block, this value will differ from both values held in the first two blocks. Thus there are two consistent copies of the new root and the update operation can continue.

This algorithm differs from the mechanism for providing stable storage used in PS-algol and Napier88. Integrity of an update of a root is preserved using a variation of Challis' algorithm [8, 11]. The root is composed of two root blocks. Held within each root are date stamps which differentiate between the active store and the shadow store. Updates occur by assigning pages in the shadow store. When the time comes to commit the changes to non-volatile store, the shadow root is overwritten with details of the update status of the shadow store. The root is flanked by two identical timestamps. The new timestamps are generated and assigned in a single write operations to the root. The two timestamps are then checked and if they are consistent then the shadow store becomes the non-volatile store and the old non-volatile store becomes the shadow store.

Both mechanisms provide adequate support for consistent and recoverable updating of a single root. The approaches can be applied to incremental updating, since the pset structure underlying Ten15 can be produced from multiple updates without affecting the root. Similarly, the use of two filestores in conjunction with Challis' algorithm allows the shadow to be incrementally built up with changed pages. The latter is inconvenienced by the need to partition into two identical halves; whereas the Ten15 mechanism means that only those blocks that have been changed need to be appended to the filestore. However the immutable nature of the filestore means that this flexibility is paid for by the need to perform off-line garbage collection.

2.6. Persistence and types

The Ten15 type system allows persistence to be arbitrarily combined with other types and type constructors to provide an orthogonal persistence mechanism. The type system also permits persistent variables to be declared as complex types. Psets are treated as a basic type by Ten15. Figure 4 demonstrates a typical mode declaration involving persistent values (*persist* ...) and persistent variables (*pvar* ...).

```
Modes:
    p1 = persist Int;
    pv1 = pvar Int;
    poly_mode = λ(X: ptr X);
    p2 = persist poly_mode[Int];
    pv2 = pvar poly_mode[Int]
```

Figure 4: Declaring persistent types.

2.7. Comparing persistent paradigms

Ten15 provides a different mechanism altogether in comparison to that found in Napier88 and PS-algol. Whereas both of these languages exhibit intrinsic persistence, where the movement of data between and main and persistent store is transparently managed at run-time. Ten15 provides a lower level approach to persistence in that the movement of data between stores is apparent within the language.

Providing persistence as operations of a language, rather than an intrinsic property, makes Ten15 a more flexible alternative, since the user is allowed to decide what data persists and at what particular instant of time. Thus, the user constructs data structures specific to his application which possess the expected performance characteristic and persistence.

Ten15 is capability based, making χ its closest related language [18]. Both languages refer to persistent data as capabilities which uniquely identify the data for its lifetime. One difference is that χ provides a mechanism for allowing the virtual update of a persistent capability without bringing the data explicitly into memory, using the *THRU* command. In addition, object creation is managed implicitly. Thus a new persistent capability is created by assigning a value to an undefined untyped persistent capability. In Ten15, this must be achieved by incremental application of persistent and variable creation operations, the types of which must be predetermined and consistent. Where they differ greatest is in the complexity of the capability generated: χ generates 128 bit capabilities which contain information concerning the type, size and, most importantly, the access capabilities for the language. Thus capabilities can be defined to be read only or writeable. Ten15 capabilities do not include such information since the persistent store is divided up into the two kinds of access modes: read only, through persistent values; and read-writeable through persistent variables. Thus all items of data can have three possible residencies: in mainstore; read only on persistent store; and as a persistent variable.

2.8. Implementation

In its current implementation, Ten15 exists as a cross-compiled system on a DEC VaxStation. A VMS file is used to model the Ten15 datastore, which the Ten15 kernel interprets as contiguous Ten15 datastore blocks of 512 bytes. As values persist they are added to the next free block as indicated by the freelist.

There is a mainstore buffer, of 32K bytes, which holds all data being passed between mainstore and datastore. The top half of the buffer is used for writing values and the bottom for reading persistent data. When either end extends to occupy more than half of the buffer, the whole buffer is flushed.

As data is added to the persistent datastore, available space decreases. As the datastore becomes full, off-line garbage collection is employeed which rids the datastore of all values, variables and psets that can no longer be referenced from anywhere with the datastore data structure.

2.9. Summary

In this section, we have presented the persistent features of the language Ten15. Ten15 is minimal in it approach to persistence in that it provides no support for transparent manipulation of persistent data; nor for controlling access characteristics of the data stored. Both facilities are functions of the language: the former provided for by operations to save and retrieve

data, and to structure the persistent store; and the latter through the separation of read only and read and writeable data into distinct type extensions. We feel, however, that the simplicity of the Ten15 mechanism is compensated for by the expressiveness of its type system, which we will discuss in greater detail in the next section.

3. Filestores and Ten15

The previous section introduced the Ten15 persistence mechanism. In this section, we concentrate on the area of filing system support and how Ten15 can be applied to such an area. A filing system provides a means for mapping names onto areas of data on a physical device, such as a disk. In traditional operating systems, such as Unix, this consists of mapping a textual name onto disk aliases which represent blocks of the physical store; in a more modern distributed system, such as Mach [25], a dedicated naming service maps names onto object addresses which are logical aliases for disk addresses.

We can identify three key elements of a filing system:

- Naming – names map onto stored data; for more general use, names should be complex or multilevel to allow a number of name spaces and users. This provides names with a degree of local scope dependent on the current depth of accessing within the file system.
- Access rights – stored data should be protected from internal and external interference. Thus a user should be able to make his file read only, writeable, executable and also place these restrictions separately on other users, or groups of users.
- Typing – the type of a value stored should be preserved along with the data. This corresponds to the notion of persistence. Unfortunately, the majority of file systems do not preserve the type of values stored: Unix treats all data orthogonaly as vectors of bytes, and such modes that it possesses, such as executables, are dependent on the ability to dynamically execute the code without failure.

Ten15 provides two out of the three criteria as basic features of the language. Access rights to persistent data are catered for by the distinct read-only and read-write capabilities of persistent values and variables respectively. The storage of typed data between program invocations is a basic facility of the persistent mechanism. The one area in which Ten15 is deficient is in the distinct naming of objects. Persistent data is represented as capabilities which incorporate a unique name, based on the disk location. In the case of persistent variables, these are relative offsets within a structured acyclic graph, assignable through a single root location. Thus if data is stored within the graph structure, the capability to retrieve it is delivered and the capability acts as the name for the object. This leads to two problems: the persistent capabilities can only be generated in the process of actively storing persistent data, so that the ‘names’ of persistent object can be generated dynamically to retrieve previously stored data. This expands into the second problem that the returned persistent capabilities must be stored between program invocations so that references to the the persistent data themselves endure. Thus the references must be made persistent between program executions which means there is the problem of circularity, since a persistent reference is required to hold onto the original persistent reference in order to later access it.

The problem of circular storage of reference is insoluble unless some form of agent is introduced which maps persistent references onto a concrete name. The address of the central repository for such names is known to all processes in the system, and so the mapping of name to value will endure for the lifetime of the name in the name space. This feature is called a

dictionary data structure, as found in the Flex PSE, which acts a central repository for mapping names to stored values.

A dictionary, in its most simple form, maps names onto values. For a Ten15 implementation, there are two issues which require resolution: how to unify the infinite union of persistent data types onto a single data type; and how to provide complex name spaces. We will address each issue in turn.

3.1. Unified types

Persistent data maintains the data type of the value as it is stored in mainstore. Thus making an integer persistent yields the persistent capability of type *persist int.* In the same way, a persistent string has type *persist string*. As a result it is impossible to provide a data structure which maps names onto persistent data in its basic form, since it can never be determined how much mainstore or filestore should be allocated and where the boundaries between distinct values will lie.

The solution to this dilemma is to make use of the *dynamic typing* facility of Ten15. Dynamic typing means that the type of a value is not evaluated until run-time. Then an operation is performed to determine the type of the value and deliver the actual contents of the dynamic value. In Ten15, dynamically typed values are represented as a pair of the value itself and a type representation. All dynamically typed values are of the same type, the type *Typed*. This means that names can now be mapped to a single unified type, which is statically typed.

The data type *dictionary* provides a set of operations to permit the construction, addition, retrieval and removal of name to persistent value bindings. A single set of operations can be provided due to the combination of *polymorphism* and *first class closure*. Values that are to be made persistent are still statically typed. Update operations on a dictionary are polymorphic, in that they can be instantiated to any type, and deliver a statically typed dictionary structure. Retrieval and assignment operations need to access the dynamically typed value to deliver the value. The dictionary data cannot export dynamically typed values since, in a reassignment operation, the stored value can be overwritten with a value of a different type. Thus these operations require as a parameter a first class procedure which evaluates a dynamically typed object and delivers its value part. The procedure is best given as a first class closure, a means by which efficient higher order functions can be implemented, in order for a clear operation to be presented and to allow a set of operations to range over a number of name mappings of the same type. A type signature for the retrieval operation follows:

```
Let dict_retrieve =
                    Ion { procedure requiring first class closure }
                    Formals R { declares free type called R }
                    (eval: TYPED → ptr R)
                    (n: NAME) → ptr R:
                    { ptr R denotes normalised polymorphic value }
```

A typical evaluation procedure, such as one to evaluate whether a dynamically typed value is of type integer, would be like:

```
Let int_eval =
    Proc
    (t: TYPED) → ptr Int:
```

Let v = TypedIsInt(t) In v Ni

The complexity of the values held in the dictionary structure is in fact more complicated than initially defined. If values are passed in as simple polymorphic values, they could be of any size. Thus, when the dictionary itself persists, it could be large and inefficient. Hence, when a value is added to a dictionary, it must provide some indication of its size, complexity and its expected access characteristics. We can map the three persistent type extensions directly onto these characteristics. If a value is small, i.e. smaller than the size of a persistent capability, it can be stored as a normal mainstore value and be persisted when the pvar which holds the dictionary is traced through; if a value is large and read-only, it can be made into a persistent value, in order to limit the amount of tracing required when the dictionary is itself made persistent; and writeable large values can be stored as persistent variables, in order to avoid having to update the dictionary and allow for object sharing. In reality, the type of each value stored is a variant of the three possible persistent extensions,i.e:

Value = (TYPEREP×union(TYPED, persist TYPED, pvar TYPED))

The *TYPEREP* value is include so that the type of the stored value can be determined without having to make a further access to the persistent store.

3.2. Complex naming

Ten15 dictionaries allow polymorphic values to be used as names. The minimum requirement for a name of any type is that it can be hashed onto a numeric value, and its equivalence can be tested against other values of the type. For these criteria, a user must provide functions which map their chosen key type onto first class closures for the dictionary manipulation operations, as demonstrated by the type signature for the dictionary creation operation:

Let create =
Ion
Formals N *{ formal type for names }*
(eq : struct(ptr N,ptr N) → bool)
(hash : struct(ptr N) → Int)
(...)

This operation builds a curried function which expects the *eq* function as its initial parameter and generates a new function declaring the *hash* function as a parameter.

Using the polymorphic naming facility, we will demonstrate how a flat name space and a multilevel directory structure can be constructed using dictionaries. Also we will outline how version control can be introduced into dictionaries, by exploiting equivalence partitioning of names.

3.2.1. Single name space

In a single name space, the names used are simple, typically a vector of characters, or a string. Equivalence can be determined by matching two names, character by character. A number of hashing algorithms exist, but a simple one would be to sum each character into a single integer.

3.2.2. Multi-level name space

A multilevel name space relies on the ability to locally scope the extent of a name and to provide path expressions to allow the resolution of a name nested deep within the multi-level name. In a Unix environment, the name space is provided for by directories of values which may be stored data or other directories. Naming is resolved by the provision of an absolute path, i.e:

/usr/tim
/usr/bill/maths/fourier.c

The need to always specify an absolute path is overcome through the use of an offset which represents the current working directory. Thus when a name is resolved the offset is concatenated with the locally scoped name to deliver the required value.

We can implement such a system using dictionaries by introducing a polymorphic naming scheme which mimics the functionality of the directory structure described above. Rather than providing the notion of disjoint directory structures, we can model the multi-level filestore on a single dictionary by mapping the complex names onto values.

The complex names are modelled as vectors of strings, where the position of the name in the vector represents the depth of the complexity. A typical complex name would be:

{"usr","bill","maths","fourier.c"}

where the depth of complexity increases from left to right. The equivalence of two such complex names depends on every component of the vector being equal. A likely hashing function is to concatenate all strings in the complex key and to hash the concatenation in the same way as the previous section.

We can provide a notion of a *current working directory* by binding a variable into the hashing and equivalence functions that are closed into the dictionary operations. Thus for each dictionary we create a persistent variable which holds a complex name which holds the base extension for the current directory. The pvar is closed into the hashing and equivalence functions and also into a set of assignment functions, such as *set* which sets the current directory, *cd* which moves up a level in the structure and *cdback* which moves down a level. Since the pvar provides a mechanism for object identity, changes to the current level of the directory result in the update being propagated to all operations which reference the value. In the case of equivalence and hashing functions, the pvar is dereferenced and concatenated to the locally scoped name. If the length of the current working directory concatenated with the local name exceeds the maximum depth for the directory structure, which is held as an environment variable within the dictionary data structure, then it is assumed that the local name is in fact an absolute path and is treated as such.

The advantage of this approach is that a single dictionary can hold an entire multi-level filestore. The only constraining factor is the performance of the name resolution data structure. If in its most simple form, the dictionary is implemented as a vector of name to value mappings, the access time will be of order O(n/2). If we use accelerated data structures, such as extensible hash tables [16], we can bring access time down to O(1). The scalability of such structures to containing numbers of complex values in excess of 10 000 has been shown in [5].

3.2.3. Version control name space

A version control filestore allows the history of a filed data structure to be preserved. This means that there are a number of versions of the data structure, each at a different stage of update. The set of versions are addressible through as a single name and individual versions appear as an extension onto this name. Such a system is available on the VAX/VMS operating system.

Thus we have a two level name space, where the name for the data structure defines the group and the version controller refers to the specific instance. One solution to this problem would be to adopt the multi-level naming approach described in the previous section. The problem with such an approach in this context is that often we wish to add new versions with the same group name, rather than having to specify the exact version extension to the name. To cover this feature, we must provide a more flexible name resolver which allows for both levels of name to be refer to stored values.

We model a version name as a pair of the group name and a name extension, such as an integer, i.e:

String×Integer

The name is hashed on the first attribute of the pair and multiple versions of the name invokes rehashing. When a new version is added, the weak equivalence of the the first field of the name is tested. The number of positive tests for weak equivalence forms the basis for a count which is the current number of versions. The value is stored and the name is extended with an incremented version count. In addition, the last versioned value is converted from a persistent variable to a persistent value, if it is stored as a read-writeable value, so that the old version is not overwritten by the new version. When a particular version is required, the user supplies the complete key which holds the group name and the version extension and an exact equivalence operation as a closure. In effect, the operations on a dictionary are doubled as a set of operations is created by the closure of a weak equivalence function, which matches only on group name, and another set which delivers single versions of a group, through the closure of an exact equivalence operation.

We can provide further operations on dictionaries which allow the removal of an entire group, the generation of a new dictionary from a group and the purging of a group up to its most recent version. These are trivially implemented on top of the basic dictionary manipulation operations.

The Ten15 immutable mechanism fits in well with this style of version control, because on every update, the entire persistent value must be rewritten. Thus the old persistent address goes out of scope. However, the name space for the version controller can bind the previous persistent value to the most recent name, so that the old version remains in scope and be used at a later date if required.

3.3. Controlling access rights

In the dictionary system we have previously described, criteria for a filestore are met: the preservation of type; and the provision of a complex name space. Access control is solved in part by the dictionary structures described since values cannot be assigned if they are stored as persistent values. However, the security of the system is problematic since any user who possesses the persistent variable capability for the dictionary can access any value held in the dictionary, since the resolution of dynamic types is available to all users and structural type

equivalence means that users need only guess the required format of the data structure rather than the name.

A solution to this problem is to restrict the domain of a dynamic type to those users which possess a password capability. This capability contains an unforgeable representation of the type of the stored value. In Ten15 there is a mechanism for constructing *conditional existential quantification*†. Existential quantification [10] is a mechanism for providing information hiding by placing a complex value behind an abstract interface. The mechanism provides structures that are akin to abstract data types. Thus we can have an existential type:

∃x.(x,x→Int)

where *x* is a value of any type, and *x→Int* is a first class procedure which takes a value of type x and delivers an integer. The type specification for the existential type is equivalent to an abstract data type.

Conditional existential quantification is a weaker form of existential quantification since the representation contained within an existential can be determined by providing the right key. Wotsits have the static type:

Wotsit = ∃x:(unique x, x)

where *unique x* is a unique capability of the same type as the hidden value. When a Wotsit is created, a unique value of the same type as the stored value is generated, or reused, and stored. The Wotsit can then be passed around as a single statically typed value, with the type defined above. When the value of Wotsit is to be retrieved a conditional matcher operation is applied to the existential value with a unique capability as the other parameter. If the parameter was generated from the same invocation of *MakeUnique{x}*, then the value will be delivered. Note that the equivalence of the type of the stored unique and parameter unique is not enough to resolve the conditional match. The following Ten15 code demonstrates the generation of a Wotsit and the conditional retrieval of the contents of the value.

Modes:
Wotsit = ∃(X: struct(unique X, ptr X))
{ ptr X defines a polymorphic value of
uniform size }
Ten15:
Let a : ptr Int = Pack 5
Let capability = MakeUniqueInt()
Let ext = ToExists:struct[Int,Wotsit](capability,a)
{ Witness type instantiated to int }
...
Let res = Assert as [Int] Match{(ext,capability) | Fail}
{*Match fails if capability does not equal unique held in ext}*

Using Wotsits, we have a means to control the access rights of users to values within a dictionary. Not only must the user have the persistent variable capability of the dictionary, he must have a copy of the unique generated along with the Wotsit in order to access a particular value on the persistent store. The type capability must be stored externally to the dictionary

---

† Instances of conditional existential quantification are affectionately known as *Wotsits.*

structure, such as a closure to a first class procedure.

In effect, Wotsits can reside alongside Typed values in the dictionary data structure, since they are both statically typed, and a union can be used to distinguish between the two types at run-time. Typed values will still be required in the case when values need to be universally available.

3.4. Dictionary operations

Dictionaries provide a suite of operation for the storage and retrieval of single values. The data type also has support for operations to alter access modes, provide views over the dictionary structure and to manipulate name bindings, such as to rename the name bindings for persistent data and to copy persistent data to another name binding. In this latter case, we have a problem concerning how multiple copies of a single persistent variable affect the update consistency. A copy operation must determine whether it is creating a new copy of the value or is merely providing an alternative name binding for the value.

In the case where each copy spawns a new version of the copied data, a new persistent variable for each name binding must be generated. If a change to a single name should be reflected in all other name bindings, then the persistent variable must be the same for each binding. This produces further complications if the access type of a stored value needs to be changed from writeable to read only. This can only be performed on a single name binding since there is no means of determining what other names also refer to the value. A means to overcome this is to create a 'sink' which holds the persistent variable when a copy operation is performed. The sink is given an unique hash number and stored in the dictionary. The value stored under the old and copied name bindings is now composed of a pair of the hash value and a unique, which distinguishes the value from a normal integer. The unique and the hash value act as a *safe pointer* [5] within the confines of the indexing mechanism of the storage structure. When the type resolution is invoked on this value, it will fail. The failure can be caught and a type resolution can be made to determine whether or not the value is being used as an indirection into the dictionary. If it is then the new hash value can be retrieved, otherwise a failure is delivered in the usual manner.

3.5. Evaluation

The dictionary data type we have described provides a flexible name to value binding mechanism for the storage of arbitrarily complex persistent values. The development of such a data structure is dependent on the low level nature of the persistence mechanism and the expressiveness of the type system, in particular in the areas of dynamic typing, existential quantification, polymorphism, higher order functions, object identity and first class functions. We can identify six areas in which the Ten15 persistence mechanism and expressive type system have combined together to the great advantage of the dictionary data type:

- Complex storage and naming systems, such as multi-level and version control stores, can be built on an essentially flat file space.
- A number of dictionaries can be constructed from a single dictionary data structure. This can be achieved through the use of polymorphism which allows the dictionary operations to be instantiated to a particular set of access characteristics, yet still maintain a uniform static type. Hence we could provide a number of hashing and equivalence protocols to the dictionary operations and have all these dictionaries residing on data structure.

- Instead of needing to export a number of references to persistent data from a program, only the root of the dictionary must be passed out. In addition, there is only one reference to each value in the data structure, making the job of garbage collection easier to implement.
- The structuring of the persistent datastore is hidden beneath the abstraction of the dictionary structure. Thus the dictionary can follow the construction of psets, ensuring that not too many pvars are added to them, so that they do not become unwieldly to manipulate. Also, pvars which are to be frequently updated can be deliberated stored higher up in the acyclic structure in order to improve performance. Thus the management of the persistent store becomes heuristic rather than arbitrary.
- The performance of a dictionary is greatly increased by the benefits of abstraction, such as highly complex algorithms, as described in [6], can be used to provide the maximum level of performance. In real terms, a dictionary structure constructed using a variation of the extensible hashing algorithms allows 10 000 name to value bindings to be stored with a guaranteed upper bound of access time of 1 second. This compares favourably with a typical user's name space which contains on average 2 000 name bindings.
- The access rights over what user can access any values in the dictionary can be achieved through the use of a dynamic typing mechanism which adds an extra dimension to the notion of a type representation by making it dependent on the exact time when it was created.

4. Conclusion

In this paper, we have presented the notion that persistent programming languages can be used as the basis for the development of standard, yet complex, operating systems facilities, such as filing systems. To this end, we introduced the persistent programming language Ten15, describing in detail the persistent programming facilities of the language, drawing out the distinct features of the mechanism in comparison to χ and Napier88.

The resulting dictionary data type provides the functionality of a filing system and could be made part of an operating system constructed using Ten15. In a sense, the dictionary data type is equivalent to the *environment* feature of Napier88 [13, 14], since block structured name bindings can be applied to arbitrarily complex persistent values. Our mechanism differs in two main ways: Napier88 supports the transparent movement of data and update of name bindings; in Ten15 this must be performed explicitly. The dictionary mechanism we have described is more flexible in what it allows as a name binding for a persistent value; hence it is more akin to an environment constructor for a mechanism such as that found in Napier88 rather than the mechanism itself.

The dictionary data type has demonstrated how the flexible manipulation of persistent data can be coupled with a highly expressive type system to permit the modelling of highly efficient complex operating systems abstractions. It is our belief that, as persistent programming languages become more efficient, they will become the systems development languages for operating systems of the future.

5. References

1. R. Agrawal and N.H. Gehani, ‘‘ODE (Object Oriented Database and Environment) : The Language and Data Model’’, pp. 36-45 in *ACM Sigmod Proceedings of the International Conference on the Management of Data* (1989).
2. M. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott and R. Morrison, ‘‘An Approach to Persistent Programming’’, *B.C.S. Computer Journal* **26**(4), pp. 360-365 (November 1983).
3. M. Atkinson, P. Buneman and R. Morrison, ‘‘Binding and Type Checking in Database Programming Languages’’, *B.C.S. Computer Journal* **21**(2), pp. 99-109 (April 1988).
4. M. Atkinson and R. Morrison, ‘‘Persistent Systems Architectures’’, pp. 73-109 in *Proceedings of the Persistent Object Systems Workshop*, ed. J. Rosenberg and D. Koch, Newcastle, Australia (1989).
5. T.D. Blanchard, *Probe: an efficient database language* , Department of Computer Science, University of York, York, UK. (1992). DPhil Thesis. In preparation.
6. T.D. Blanchard and J.A. McDermid, ‘‘The Probe Project’’, YCS 167, Department of Computer Science, University of York, York, UK. (January 1992).
7. M. Brandreth, P.W. Core, I.F. Currie, N.E. Peeling, M. Stanley and J.M. Foster, ‘‘Ten15 Prototype’’, R.S.R.E. Report 91025 (1991).
8. A.L. Brown, ‘‘Persistent Object Stores’’, Persistent Programming Research Report 71, Universities of Glasgow and St. Andrews Computer Science Departments (March 1989).
9. L. Cardelli, *Amber*, AT&T Bell Laboratories, Murray Hill, New Jersey (1984).
10. L. Cardelli and P. Wegner, ‘‘On Understanding Types, Data Abstraction and Polymorphism’’, *ACM Computing Surveys* **17**(4), pp. 471-522 (December 1985).
11. M.P. Challis, ‘‘Data Consistency and Integrity in a Multi-User Environment’’, in *Databases: Improving Usability and Responsiveness*, Academic Press (1978).
12. I. F. Currie and J. M. Foster, ‘‘An Evaluation of Flex Programming’’, R.S.R.E. Memorandum 86003 (1984).
13. A. Dearle, *On the Construction of Persistent Programming Environments*, University of St Andrews, Department of Computional Science, North Haugh, St Andrews, KY16 9SS (1988). PhD Thesis.
14. A. Dearle, ‘‘Environments: A flexible binding mechanism to support system evolution’’, Persistent Programming Research Report 67 , Universities of Glasgow and St. Andrews Computer Science Departments (1988).
15. ‘‘The PS-algol Reference Manual, Fourth Edition’’, Persistent Programming Research Report 12, Universities of Glasgow and St. Andrews Computer Science Departments (1987).
16. R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, ‘‘Extensible Hashing - A Fast Access Method for Dynamic Files’’, *ACM Transactions on Database Systems* **4**(3), pp. 315-344 (September 1979).
17. J. M. Foster, ‘‘The Algebraic Specification of a Target Machine: Ten15’’, in *High Integrity Software*, ed. C.T. Sennett, Pitman (1988).
18. A.J. Hurst and A.S.M. Sajeev, ‘‘A Capability Based Language for Persistent

Programming: Implementation Issues'', pp. 110-125 in *Proceedings of the Persistent Object Systems Workshop*, ed. J. Rosenberg and D. Koch, Newcastle, Australia (1989).

19. S. Khoshafian, ''A persistent complex object database language'', *Data and Knowledge Engineering* **3**(4), Ashton Tate, Walnut Kreek, CA, USA (February 1989).
20. A.M. Lister, *Fundamentals of Operating Systems*, Macmillan Computer Science Series (1984). Third Edition.
21. D.J. McNally and A.J.T. Davie, *Two models for integrating persistence and lazy functional languages*, University of St Andrews, Department of Computional Science, North Haugh, St Andrews, KY16 9SS (1991).
22. R. Morrison, F. Brown, R. Connor and A. Dearle, ''The Napier88 Reference Manual'', Persistent Programming Research Report 77, Universities of Glasgow and St. Andrews Computer Science Departments (July 1989).
23. N. E. Peeling and K. R. Milner, ''KeepSake: A Database Kernel'', R.S.R.E. Memorandum 88014 (March 1989).
24. J.E. Richardson and M.J. Carey, ''Persistence in the E language : Issues and Implementations'', *Software - Practice and Experience* **19**(12), pp. 1115-1150 (December 1989).
25. F. Vaughan, T. Schunke, C. D. Marlin, A. Dearle and C. J. Barter, ''A Persistent Distributed Architecture supported by the Mach Operating System'', *Proc. First USENIX Workshop on the Mach Operating System*, Burlington, Vermont, pp. 123-140 (October 1990).
26. F. Wai, ''Distributed PS-algol'', Persistent Programming Research Report 60, Universities of Glasgow and St. Andrews Computer Science Departments (1988).
27. C. Zaniola, ''The Database Language Gem'', *ACM Sigmod Record* **13** (4), pp. 207-218 (December 1983).